Departament d'Enginyeria Informàtica i Matemàtiques

GroupCast – *Total Order* Multicast and *Group* Communication

Task #2: **Lamport clocks, total ordering, failure tolerance, leader election**

Distributed Systems 2016/17

The single biggest problem in communication
is the illusion that it has taken place.
George Bernard Shaw

The goal of the second homework is to implement a minimalistic group communication library that provides Total Order Multicast (TOM) to a number of participants via the PyActor library [1]. TOM is a very basic primitive of distributed systems. For example, a group of replicated files cannot be in the same state unless all replicas apply the updates in the same order. In these situations is where TOM takes over. In this task, we will compare two different implementations of TOM. The first implementation is based on the use of Sequencers. The second implementation is built upon the famous paper from Lamport entitled "Time, Clocks, and the Ordering of Events in a Distributed System" [2].

1. Group Communication

The first task is to create a Group Communication (GC) library that implements a Group Membership Service (GMS) keeping track of users participating in the group. This library should offer a basic API with the classical functionality: `join`, `leave` and `get_members`. To implement the `Group` actor service, you might leverage the `Tracker` actor from Task#1. Provide here a fault-tolerance mechanism to detect peers that fail without explicitly calling the `leave` method.

2. Total Order Multicast (TOM)

In TOM, every member of the group is required to deliver all messages sent within the group in identical order, which does not necessarily correspond with the real time at which these messages were sent out.

Notice that as messages may be received in any order, many of the existing TOM protocols make use of a queue to buffer out-of-order messages until they can be delivered to the application in the same total order for processing. This implies that every peer i should maintain a queue Q_i (initially empty) to which messages are appended as soon as they are received, which is used to reorder events locally before delivering it to the application.

We will consider two different implementations of TOM with the following common API:

1. `multicast:` This method will be used by a peer to send a message to the rest of peers in the group.
2. `receive:` A peer i will call this method to transmit a message to another peer j.
3. `process_msg:` This internal method will be called by a peer i only when it is safe to deliver a message to the application for processing, i.e., when the message at the head of the queue Q_i preserves total order.

2.1. TOM with a Sequencer

The idea here is to implement total ordering using a sequencer such as described in Colouris's Distributed Systems book. Essentially, a sequencer is a process that assigns a unique timestamp `seq` to every message `m` that it receives, and multicasts it to every other member of the group as shown in:

```
{The sequencer S}
define seq: integer (initially seq=0}
do receive m → multicast (m, seq) to all members;
   deliver m;
   seq := seq+1;
od
```

To implement the TOM API, you can leverage your `Peer` implementation from Task#1. However, one important difference here is that in our case the sequencer will not multicast messages. Multicast will be done by the peers in the group: The method `multicast` will send a message to all peers in the group with the timestamp provided by the `Sequencer` actor. That is, the `Sequencer` actor will only assign unique timestamps to multicast requests. The messages will be sent in correct order to the `process_msg` method. Messages received out of order will be buffered until the message with the expected number arrives.

Since a single sequencer is a single point of failure (SPOF), it is also necessary to provide a failure tolerance mechanism as part of this task. One simple solution would be resort to the bully leader election algorithm to elect a new sequencer when it fails, and thus, overcome the SPOF problem

2.2. TOM with Lamport Clocks

One criticism of sequencers is that they can become a bottleneck. To address this issue, there exist several distributed implementations of TOM that do not use a central process as a sequencer. One of the most famous implementations was proposed by Lamport in his seminal article entitled "Time, Clocks, and the Ordering of Events in a Distributed System" [2]. Subsequent works built upon his protocol, and propose new TOM algorithms.

First of all, we recommend you to take a look at the section called "Ordering the Events Totally" in the Lamport's paper [2]. Once you have understood the implications of ordering events totally, your task is to leverage your `Peer` implementation from Task#1 to implement a variant of Lamport's algorithm, which has been already document in the book "Distributed Systems: Principles and Paradigms" by Tanenbaum and Van Steen. Concretely, the algorithm to implement will be the following:

1. Each message `m` is always timestamped with the current Lamport clock of its sender (when a message is multicast, it is conceptually also sent to the sender).
2. When receiving a message `m`, the receiver will it put into a local queue Q, ordered according to its clock. Then, the receiver will multicast a timestamped acknowledgment (ACK) to everybody. Note that this means that the timestamp of the received message will be always lower than the timestamp of the acknowledgment.
3. As ACKs and messages are received, each process will order them in the local queue Q according to their clock value (and process number).
4. A process will finally deliver a queued message m to the application it is running only if: $1)$ m is at the head of the queue Q; and $2)$ it has been acknowledged by each other process. At that point, `m` is removed from Q and handed over to the application;

In terms of our TOM API, the method `multicast` will be used to send a message to all peers in the group. Messages will be kept in each peer's queue until all peers confirm the reception with timestamped ACKs. Messages confirmed by all peers will then be processed by the application by calling `process_msg` in total order.

3. Requirements

The project will be implemented using PyActor [1]. The specific requirements for this project are the following:

1. Create a group of n peers that enter a group and begin to multicast messages. n is a parameter and can change in each simulation run.
2. Check that all peers receive the events in the same order in any of the implementations. You should provide a clear means to demonstrate that the events are applied in the same order in all the n peers. One solution is to create a `Monitor` actor or that every peer writes to a separate file the order of event delivery.
3. Before running total order multicast, the n peers should first join the group. In the multicast phase, we will assume that there is no `leave` operation incurred by peers.
4. To force the reordering of multicast messages, you should artificially introduce delays by using the `sleep` function.
5. It is also MANDATORY that all the implementations work in two separate terminals.

4. Tasks (Weights)

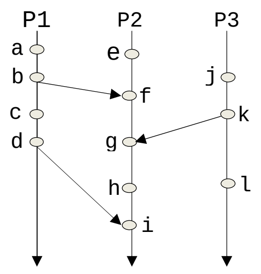
1. **(30%). GMS.** Implementation of the GMS Service. Provide the appropriate tests to check that the GMS service is working fine.

2. **(30%). TOM with a sequencer.** Correct implementation of TOM with a sequencer. This includes the implementation of the bully leader election algorithm seen in class.

3. **(15%). TOM with Lamport clocks.** Correct implementation of the distributed TOM protocol with Lamport clocks.

4. **(20%). Theoretical Questions.** Also, you should answer to the following questions:

    a. *Lamport defined a notion of logical time based on event ordering – the so called "happened-before" relation. Show how a global total ordering of events can be achieved based on this notion of Lamport's logical clock by specifying the Lamport clocks for each event in the next figure.*



    b. *How do vector-clocks extend the notion of Lamport's logical clocks?*
    c. *State the leader election problem and tell the communication cost for the Bully algorithm. Assuming only crash failures, which is the minimum number of participants that are needed for an election to be successful?*
    d. *Compare both communication algorithms in terms of communication cost, fault-tolerance, and scalability.*

5. **(5%). Personal github.** All groups must create their own github repository with their code and documentation. If possible, we recommend including at least code coverage and health links. This requires the development of unitary tests for your code. Example of a https://github.com/danielBCN/PyActor-example

5. Deliverables

You will have to submit the **answers to all the questions**, the **final design document** as well as the code itself in any format (preferably **.pdf** and **.doc**); the allowed languages are Catalan, Spanish and English. The **final design document** should include a concise explanation of your design choices of about **3**-page length, implementation details, as well as what tests you have run to confirm that your implementation of *GroupCast* works as specified. Do not forget to include **the source code** written in Python when submitting your solution.

Last available date: May 26, 2017;

□ After the evaluation of the task, the professor responsible for your group may ask you about some points of your delivery to assess your degree of knowledge.

Final Hint: We recommend you to do not procrastinate and take advantage of the class hours in the laboratories.

Note: In case you find bugs in the code or the instructions, email *marc.sanchez@urv.cat* and I'll fix the issue, post revised versions on the website and inform everyone.

6. References

[1] PyActor library, 2017, https://github.com/pedrotgn/pyactor
[2] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", http://amturing.acm.org/p558-lamport.pdf